THE ROBOT

event

 waits for the completion, office Navigation

Your Robotic Rover is the first of its kind. It has

been used in a variety of settings. In offices and

the conference room. The robot is programmed to

move around the room, respond to commands, and

assist with various tasks. The robot is designed to

be user-friendly and efficient in its operations.


KANSAS STATE UNIVERSITY

Robot Software

David A. Cutkosky

Kansas State University
base can rotate independently. The robot can translate forward and backward on its three wheels, which rotate together. The robot has a zero turning radius and a maximum speed of 20 inches/second. The Nomad 200 robot has a simulation system that runs on UNIX workstations. This simulation system allows extensive execution of the software in a variety of situations.

The programming was done in C++ under the Linux operating system that was installed on the 486 PC on board the Nomad 200. Although the team used a workstation and wireless ethernet to initiate the execution of the robot software, the Nomad 200 was run as an autonomous robot under the control of the software on board the robot.

The Design

The object-oriented approach provided robust software to control the robot. The main principle of the software development was to partition the tasks and responsibilities into top, middle, and low levels. For example, the top level needed to know about what the Office Navigation event was and how to solve it but did not need to know about obstacle avoidance. The behaviors at the bottom level needed to worry about low-level responsibilities, such as avoiding obstacles and not hitting walls but did not need to know about the overall strategy for solving the task. Figure 2 shows a simplified version of the object model. Most of the attributes and some of the methods are not included. The classes marked with an asterisk run as separate threads.

The top-level object high_level_control was responsible for the sequencing of the path-planning, high-level motion, observation, estimation, and direction tasks and for high-level error recovery. The high_level_control object handles the direction of motion down hallways and into and out of rooms, but lower-level objects control the local movement.

Below the top level are objects to handle the mapping functions, the detection functions, and the robot motion. The midlevel-object map is responsible for the mapping and path-planning functions.

The midlevel object detect is responsible for using the camera to detect motion in the conference rooms. The low-level-object camera controls the actual camera. In each room, the camera was aimed in three directions, overlapping the whole area of the room, and in each of the directions, a sequence of camera images was compared by the detect object to identify any movement or change in the image.

The last midlevel object, motion, is responsible for general hallway motion, the exiting of rooms, and the entering of doorways.

The move function in the motion object does two important tasks (figure 3). The first task is to avoid moving objects. It calls the function blocked to check on the immediate area in the direction of motion. If there is an obstacle to forward motion, it waits and retries. If there is not an obstacle or if the obstacle has moved, it continues to monitor the areas directly in front of the robot. It plans the local movement necessary to avoid obstacles. It also uses a low-level object, vector, to identify the immediate free area in front of the robot, allowing the robot to smooth out the motions necessary for moving through doorways and around obstacles. The team's software controller was nicknamed slick willie because its motion through doorways was a graceful sliding motion instead of the boxy movements common in robots centering on, and moving through, doorways.

The low-level object vector is responsible for maintaining the local view of the surround-
movement control
caused by all the functions that do middle-level
operations on streams of raw data. Movement blocked is
forewarned by the middle-level movements. If there is a
potential of middle-level events, the high-level module
reacts to detect possible events. If the event is detected
an alarm will go off and the robot is stopped. The
alarm also stops any movement. The alarm also
causes the robot to change its direction.

In addition, the robot can detect walls and obstacles
based on the data it receives. When an obstacle is
detected, the robot will slow down and change its
direction to avoid the obstacle. If the obstacle
continues to move, the robot will stop and
revert to its original path.

Figure 2: Simplified Object Model of Control Software

The system is designed to be modular and
scalable, allowing for easy expansion and
adaptation to new situations. The system
uses a combination of hardware and
software components to achieve its
goals.

node

motion
dead

getSensor

getCamera

buildMap

map

simply will follow a line of, say, 10 degrees
around a wall, and so on. It is flexible in
deciding the location of objects and
inferences. It uses a graphical interface
of the

Articles
while (1) {

    // see if intended path of travel is blocked
    if (blocked())
        wait a few seconds - say "get out the way"
    if (blocked())
        return (failure);

    if (left_opening()) and (in_left_window()) and (looking_left)
        return (success);

    if (right_opening()) and (in_right_window()) and !looking_left)
        return (success);

    // forward speed - based on a linear function dependent upon how close to obstacles
    forward_speed = get_forward_speed();

    // get the direction to go - this will always be a vector toward the best "freespace" area in
    // the intended movement direction
    turning_angle = vector->get_vector();

    // the turning speed is a linear function based upon how far we have to turn
    turning_speed = get_turning_speed(turning_angle);

    // enqueue the move
    low_level_control->enqueue(vm, forward_speed, turning_speed, 0);
}

Figure 3. Algorithm for Move Function in Class Motion.